記　号　処　理 53－2
（１９８９．１１．２２）

マウスインタフェースのための付加機構について

古坂孝史†、井田昌之‡
†青山学院大学/CSK、‡青山学院大学

Lispウィンドウシステムでのマウスイベントの扱いについて述べる。非同期的なマウスイベントの処理の可能性を探るために、既存のCommon Lisp処理系からSun Common Lisp, Allegro Common Lisp及びKCLを選び、マウスイベント処理の評価実験を行なった。Sun Common LispやAllegro Common Lispでは、マルチプロセス機能により非同期的なマウスイベントを利用者に提供できるが、KCLそのものの機能では、マウスイベントを取り込めても、非同期的には行なえない。この機能を実現するために、X-window上のKCLに対し、Xクライアントとしてマウスセンスを行なうプロセスと、そのプロセスからマウスイベントを割り込みとして受け取る機構を試作した。割り込み機構のKCLへの付加はKCLのコンソール割り込み処理を拡張することで行なった。

A Provision for A Mouse Interface

Takashi Kosaka, Masayuki Ida
CSRL, Information Science Research Center
Aoyama Gakuin University
4-4-25 Shibuya, Shibuya-ku, Tokyo JAPAN 150
† Kosaka@europa.cc.aoyama.ac.jp, ‡ ida@cc.aoyama.ac.jp

This paper describes a provision for a mouse event interface on Common Lisp implementations. Researching the possibility of asynchronous mouse event method, we choose Sun Common Lisp, Allegro Common Lisp and KCL among several Lisp implementations, and these are experimented by the mouse event handling. Sun Common Lisp and Allegro Common Lisp can cope with mouse events asynchronously using theirs multi process mechanism. But, though KCL can handle a mouse event by own facility, its mechanism isn't enough for our purpose. The provision for KCL on X-window has an event-driven interrupt mechanism and an X-client process to sense every mouse events. The event-driven interrupt mechanism is implemented by extending the interrupt facility for KCL on console interrupt.

1 はじめに

現在のCommon Lispの仕様には、グラフィックユーザインタフェースやマウス等のポインティングデバイスインタフェースの定義は無い。その為、高性能ワークステーションのユーザインタフェースは、各々の処理系に委ねられている。既存のlisp処理系は各々、ワークステーションの持つさまざまなwindow systemに対して独自なユーザインタフェース仕様を持っている。

特に、マウスイベントに対して起動される処理（マウスメソッドとここでは呼ぶ）の動作は同一の仕組みではない。

また、複数のマウスメソッドがある場合にそれらの間でのスケジューリングは異なっている。特に、本質的にはマルチプロセスでないUNIX OS上のCommon Lisp処理系の上でこれらの処理を記述する場合、既存の処理系では、各々の持ついわゆるマルチプロセス機能を利用しなければ、イベントキューが単一であるため、利用者が意図したようなマウスメソッドのディスパッチが起きないことがある。

UNIX上で利用可能なCommon Lisp処理系の内、Sun Common Lisp (Sun CL)[1], Allegro Common Lisp (Allegro CL)[2], KCL[3]を取り上げ、それらの機能／モデルについて調べ、その結果を基に各々の処理系に対して同一の機能を利用者に提供できるような共通した仕組みを模索した。その結果、Sun CL及びAllegro CLでは、それぞれの最新版ではそれぞれのマルチプロセス機能により実現できることがわかった。KCLの場合、新たに機構を設計し付加することで実現できることがわかった。この機構について述べる。KCLに対する実験は、LUNA上のX-window 11 Rel.2 、socketによるプロセス間通信そしてUniOS-B UNIXを利用して行なった。しかし、ここで述べる議論は、他のマシン上でもそのまま適用することが可能である。

2 非同期処理の問題

マウスメソッドを記述するには、一般に次のような手順をとる。

1. ウィンドウを定義する。
2. マウスメソッドを定義する。
3. そのマウスメソッドとウィンドウを関連付ける。

こうすることで、プログラムの実行中にマウスイベントが発生すると対応するメソッドが実行するような仕組みを記述できる。しかし、単純なイベント処理機構では、一つのマウスメソッドの実行中に他のマウスイベントが生じた場合、それに対応するマウスメソッドを即座に起動することはできない。

実験として、ウィンドウを作り、それに右ボタンメソッドと左ボタンメソッドを置き、それらの間で変数を共有する場合を考える。ここでは、ウィンドウを `*window*`、右ボタンメソッドを `right-button`、左ボタンメソッドを `left-button` とする。

これをAllegro CL 3.0及びSUN CL 3.0で記述すると図1のようになる。このプログラムは、右ボタンメソッドの実行中に左ボタンイベントが発生する場合のイベントディスパッチを見るものである。

`*window*` に対する `right-button` は、`format` 文を繰り返すもので、マウスイベントの発生により大規模な処理の起動を想定している。又、繰り返し処理のなかでは、グローバル変数 `*flg*` を見ており、処理実行中に、`*flg*` の値が `NIL` になれば、処理を停止する。一方、`left-button` は `*flg*` の値を `NIL` にする。そして、`*window*` でマウスの右ボタンをクリックし、`format` 文で文字列が出力されている間に、そのウィンドウで左ボタンをクリックしてみた。

プログラマの気持ちとしては各メソッドは非同期に並行して動作していて右ボタンメソッドが動作している途中でも、左ボタンイベントが発生するとその処理が並行して進むように考えることができると良い。ところが、図1のプログラムを実行させると表1のような結果となり、右ボタンメソッド終了後に、左ボタンメソッドが起動される。この表1で明らかなように、これらの処理系では各メソッドは同時に並行して動いているわけではない。あくまでシングルプロセスの範囲内で動作している。

3 非同期実行モデル

以上のような現状に対して、次のような処理モデルを実現する。

「通常のUNIXのプロセス処理の範囲内において、各マウスメソッドが、非同期にディスパッチされる」

(A) Allegro Common Lisp

```
(defvar *window*                        ;;; ウィンドウを作る
  (make-window-stream :left 10 :bottom 10 :width 200 :height 200 :activate-p t))

(defvar *flg* nil)                      ;;; *flg* の初期設定

(enable-window-stream-page-scrolling *window*) ;;; ページスクロールモード設定

(defun right-button (window-stream mouse-state &optional ev) ;;; 右ボタンメソッド
    (let ((i 0))
       (setq *flg* T)                           ;;; *flg* をTに設定。
       (dotimes (i 10000)                       ;;; 表示処理を10000回繰り返す
          (format window-stream "Hello loop ~%")
          (if (null *flg*)                      ;;; *flg* がNILになれば、処理を抜ける
              (return)))))

(setf (window-stream-right-down *window*) 'right-button) ;;; 右ボタンメソッド設定

(defun left-button (window-stream mouse-state &optional ev) ;;; 左ボタンメソッド
    (setq *flg* NIL))           ;;; *flg* をNILに設定。

(setf (window-stream-left-down *window*) 'left-button)  ;;; 左ボタンメソッド設定
```

(B) Lucid Common Lisp

```
(defvar *window*                        ;;; ウィンドウを作る
    (make-window :x 10 :y 10 :inside-width 200 :inside-height 200))

(defvar *flg* nil)                      ;;; *flg* の初期設定

(defun right-button (viewport region char x y) ;;; 右ボタンメソッド
    (let ((i 0) (height (bitmap-height viewport)))
       (setq *flg* T)                           ;;; *flg* をTに設定。
       (dotimes (i 10000)                       ;;; 表示処理を10000回繰り返す
          (if (< (stream-y-position viewport) height)
              (format viewport "Hello loop ~%")
              (setf (stream-y-position viewport) 0))
          (if (null *flg*)                      ;;; *flg* がNILになれば、処理を抜ける
              (return)))))

(defun left-button (viewport region char x y) ;;; 左ボタンメソッド
    (setq *flg* NIL))           ;;; *flg* をNILに設定。

;;; ウィンドウ内にアクティブリージョンを作り、ボタンメソッド設定
(make-active-region (make-region :x 0 :y 0 :width 200 :height 200)
    :bitmap *window* :mouse-left-down #'left-button
    :mouse-right-down #'right-button)
```

Figure 1: Allegro Common Lisp と Sun Common Lisp によるテストプログラム

Table 1: テスト結果

Lisp 処理系	結果
Allegro Common Lisp	左ボタンメソッドの起動がかからず、繰り返し処理が最後まで行なわれた。尚、右ボタンメソッドの終了後、左ボタンメソッドが起動された。
Sun Common Lisp	同上

これを実現するには、原理的には次の二つの方向がある。第一は、マルチプロセスを用いる方法である。発生するイベント全てに対して各々に対するプロセスを発生させるような仕組みである。例えば、SUN の light-weight-process は多少それに役立つことができる。第二は、割り込みによる方法である。発生するイベントごとに割り込みをかけ、イベントに対応する処理を実行させるような仕組みである。例えば、UNIX の signal や sigvec を使う方法は、これに近いといえる。

3.1 SUN CL における実現の方向

Sun CL では 3.0 よりマルチプロセス機能が入り、これによりそれぞれのマウスメソッドをプロセスとして動作させることも可能になった。そのためには各マウスメソッドを `make-process` により並行して動くプロセスとして起動する。

3.2 Allegro CL における実現の方向

Allegro CL では 3.0 よりマルチプロセス機能が入り、これによりそれぞれのマウスメソッドをプロセスとして動作させることも可能になった。そのためにはプロセススケジューラを (`start-scheduler`) により起動し、各マウスメソッドを `mp:process-run-function` により並行して動くプロセスとして起動する。

3.3 KCL に対してとった方策

KCL そのものにはマルチプロセス、割り込みやマウスインタフェース等の機能がない。しかし、KCL は、C 言語とのインタフェースを備えており、Lisp から C、C から Lisp を呼び出すことができる。これを用いてそれらを付加する。次の機能を実現する。

- ベースとなる X-window のサーバーに対して常時センスをして、マウスイベントを全て取り込む。
- 各々のマウスイベントに対するマウスメソッドを記述する。
- 記述したマウスメソッドに対して、それが対応するマウスイベントが発生すると直ちに処理の起動をかける割り込み処理機構をおく。
- イベントのセンスと割り込み処理の間の連係機構をおく。

4 KCL に対する実現

4.1 マウスイベントのセンス

X-window では、X サーバーから送られてくるイベントはクライアント側で常時センスし、クライアントは、イベントを受け取ってセンスするループから抜けイベントに合った処理を行なう。この結果、クライアント側が、イベントに対する処理をしている間は、X サーバからイベントは送られてくるが、それらを敢えてセンスしない限りイベントに対応した処理を実行できない。

従って、X-window のクライアントプログラムを KCL の C 言語インタフェースを用いて lisp で記述した場合、イベントをセンスするループを記述する必要があり、サーバから送られてくるマウスイベントに対して常時、非同期的な処理ができない。

```
(set-macro-character #\% #'(lambda (stream char) (values (read-line stream))))
(defentry lisp_c (int) (int lisp_c))

(Clines                         ;;; Lisp から呼ばれる関数 lisp_c(C 関数だけを呼び出す )
%lisp_c(x)
%int x;
%{  printf("x = %d \n",x); }
)
(defun call-from-c (x)    ;;; 関数 c_lisp から呼ばれる関数 call-from-c
     (print x))

(defCfun "c_lisp()" 1     ;;; call-from-c を呼び出す関数 c_lisp
%int x;
%     x = 10; lisp_c(x);  /* Clines により定義された lisp_c 関数を呼び出す */
     ('nil "vs[0]")
     ((cons (int x) "vs[0]") "vs[0]")
     ((call-from-c "vs[0]")))     ;;; defun で定義された call-from-c を呼び出す。

(defentry c_lisp() (int c_lisp))
```

Figure 2: KCL に置ける C 関数インタフェースの限界

そこで、マウスを常時センスする処理を、UNIX 上で独立したプロセスにした。本稿では、この処理をポーリング処理と呼ぶ。これにより、KCL で何らかの処理を行なっていても、UNIX の時分割処理により並行してマウスのセンスが行なえる。

4.2 割り込み処理機構

マウスイベントを KCL 側で取り込むためには、UNIX での signal による割り込み処理で任意の Lisp 関数が起動できるようにする。

本来ならば、この機能は Lisp と C 言語を使って利用者が記述できる方が利用者に対する自由度は大きい。しかし、KCL のもつ foreign call 機能では、実現は困難である。この例を Lisp から呼ばれる C 関数だけの lisp_c 関数と C 関数から Lisp 関数を呼んでいる c_lisp 関数を使った図 2で示す。

図 2の lisp_c は、一般の C 関数のように正しく関数名として扱われる。lisp_c は他の C 関数からの呼び出せるが、反面、lisp_c から Lisp 関数を呼び出すことはできない。関数 c_lisp の名は、正しく関数名として扱われない。この為、c_lisp は他の C 関数から関数名が見えなくなり、他の C 関数から呼び出しが困難となり、signal を利用できない。Signal はその signal が生じた時に制御を渡すべき場所のポインタ、あるいは関数名を引数として渡さなければならないからである。

そこで、別の方法を利用した。KCL ではユーザのコントロール C 入力による割り込み処理が、ソースコード中の unixint.c で記述されている。Unixint.c 中の割り込み処理を初期化している部分に、新たに割り込み処理を追加することで、割り込み処理による Lisp 関数の起動を可能とした。

割り込み処理の追加部分を図 3に示す。図 3中の *HANDLER-HOOK-1* に予め設定された Lisp 関数が、割り込み信号 2 により起動される。

4.3 イベントセンスと割り込み処理の連係 – 二つのプロセスの間の IPC

前節で述べた方法により、KCL に割り込み処理の組み込みを実現した。この拡張機能を利用して、マウスメソッドを非同期的に起動する処理を Lisp で新たに記述した。この処理を割り込み lisp 処理と呼ぶ。

割り込み lisp 処理には、ポーリング処理を UNIX でいう fork,exec する機能を置いた。KCL 上で、割り込み Lisp 処理を起動すれば自動的にポーリング処理を新しいプロセスとして生成する。

```
init_interrupt() /* KCL が起動された時、呼ばれる KCL オリジナル */
{
#ifdef 追加
    struct sigvec sv;              /* Signal 2で my_handler を起動する */
    sv.sv_handler = my_handler;
    sv.sv_mask = 0;
    sv.sv_flags = 0;
    if (sigvec (2, &sv, 0) != 0) {
        perror ("sigvec");
        exit (1);
    }
#else /* 元の処理 */
    signal(SIGINT, sigint);
#endif
    signal(SIGFPE, sigfpe);
}

init_interrupt1()       /* KCL が起動された時、呼ばれる KCL オリジナル */
{
    SVinterrupt_enable
    = make_si_special("*INTERRUPT-ENABLE*", Ct);
      .
      .
#ifdef 追加
    Vhandler_hook1 = make_special ("*HANDLER-HOOK-1*",Cnil);
#endif
}
     .
     .
my_handler1 (sig, code, scp) /* Kill(pid,2) により呼び出される割り込み関数 */
int sig, code;
struct sigcontext *scp;
{
    vs_mark;
    if(Vhandler_hook1->s.s_dbind != Cnil) {
         /* *HANDLER-HOOK-1* に 設定された Lisp 関数取り出す */
         object hookfun = symbol_value(Vhandler_hook1);
         ifuncall1(hookfun,Cnil); /* 取り出された Lisp 関数の起動 */
     }
     vs_reset;
}
```

Figure 3: unixint.c の改良

Table 2: マウスイベントの種類

マウスイベント名	意味
:button	マウスボタンが押下された。
:mouse-cursor-in	マウスカーソルがウィンドウに入った。
:mouse-cursor-out	マウスカーソルがウィンドウから出た。

各々のプロセスを、プロセス間通信を用いて接続した。その間には入力や出力を処理する為のプロトコルを置いた。このポーリング処理のプロセスが、Xサーバから、マウスイベントを受け取った時、KCL側のプロセスに割り込み信号を送る。するとKCL側のプロセスは割り込みが起こり、割り込みLisp処理で、*HANDLER-HOOK-1*に定義された関数が起動される。この関数はマウスイベントの入力待ちの処理を記述してある。入力待ちの処理は、ポーリング処理のプロセス側からマウスのイベント情報が送られてくるので、それを受け取り、その情報に対応したマウスメソッドを呼び出す。

このような割り込み処理を行なうには、KCL側のプロセスとポーリング処理のプロセスとの間で処理の同期を取る必要がある。割り込みLisp処理は、入力待ち処理が終了するまでsignalの割り込みプロテクトをかける。入力待ち処理が終了した時、ポーリング処理側にプロセス間通信で終了を通知し、プロテクトを解除する。又、ポーリング処理側では、割り込みLisp処理のプロセス側から終了を通知されるまで、マウスイベントはポーリング処理の中のqueueに溜ておき、通知された後、順次割り込み信号を送る。

5 KCLに対するマウスインタフェースの評価

KCLに対して実現したマウスインタフェースは、ウィンドウ内でマウスイベントが発生した時、起動されるメソッドを記述するものと、マウスの状態を得るものである。表2にこのインタフェースで得られるマウスイベントを示す。

表2に示されるように、アプリケーションでは各々のマウスボタンの押下した瞬間と、ウィンドウにマウスが出入りした瞬間のイベントを得られる。

ウィンドウで非同期的に起動されるメソッドを記述するには、予めメソッドの関数を記述して、これをウィンドウのオブジェクトに設定する。例えば、あるウィンドウでマウスボタンが押下された時にそのウィンドウへHello Mouseの文字列を出力させる場合は、

```
(defvar *window* (make-window-stream :left 10 :bottom 10 :width 200
      :height 200 :activate-p t))
(defun mouse-in (window mouse-state &optional event)
       (format window  "Hello Mouse ~%"))
(setf (window-stream-button *window*) 'mouse-in)
```

とする。マウスイベントに対するメソッドは3種類しか用意されていないが、起動されるメソッドの引数のmouse-stateにボタンの状態が登録されているので、各々のボタンのイベントに対する処理を分岐できる。

非同期的なマウスインタフェースを調べるために、図4のテストプログラムを作成した。このプログラムは、図1で示したプログラムと同じ処理をする。図4のプログラムを実行したの結果、右ボタンクリックで起動する繰り返すformat文による文字の出力は、左ボタンクリックで停止した。このことにより、想定したモデルが実現したといえる。

6 終りに

現在、UNIX上で良く用いられている三つのCommon Lisp処理系（Sun CL, Allegro CL, KCL）に対してマウスインタフェースの評価実験を行なった。

そして、想定した機能を実現するために、KCLに対してマウスインタフェースを作成した。このインタフェースは利用者に対する応答性能を良くすることを重点に設計した。

```
(defvar *window*          ;;; ウィンドウを作る
  (make-window-stream :left 10 :bottom 10 :width 200 :height 200 :activate-p t))
(defvar *flg* nil)        ;;; *flg* の初期設定

(enable-window-stream-page-scrolling *window*)  ;;; ページスクロールモードに設定
(defun button-down (window-stream mouse-state ) ;;; ボタンメソッド
     (let ((state (mouse-state-button-state mouse-state)))
         (case state
             (:mouse-right-1 (right-button window))  ;;; 右ボタンが押下された場合
             (:mouse-left-1  (left-button window))   ;;; 左ボタンが押下された場合
             (t nil))))
(defun right-button (window)            ;;; 右ボタン処理
     (let ((i 0))
         (setq *flg* T)                 ;;; *flg* を T に設定
         (dotimes (i 10000)             ;;; 表示処理を 10000 回繰り返す
             (format window "Hello loop ~%")
             (if (null *flg*)           ;;; *flg* が NIL になれば、処理を抜ける
                 (return)))))
(setf (window-stream-button *window*) 'button-down) ;;; ボタンメソッド設定

(defun left-button (window)               ;;; 左ボタン処理
     (setq *flg* NIL))                    ;;; *flg* を NIL に設定
```

Figure 4: KCL によるテストプログラム

その機構は、割り込み処理機構の付加とポーリング処理プロセスと割り込み Lisp 処理プロセスとの機能分担とそれらの間のプロセス間通信により実現した。なお、本稿では述べていないが、KCL に対し割り込み処理を実現し、システムとして運用する場合、ガベージコレクションに対する保護機構も必要である。本稿で示した機構は、日本語 Common Windows[4] に利用されている。

今後の展望としては、並列 Lisp もしくは子プロセスの記述／生成機構に関する研究へ進んで行きたい。なお、現在、筆者らにより YY ウィンドウ [5] を Sun CL, Allegro CL, KCL, Symbolics Genera 等の上に実現を進めているが、この実験はそのための基礎実験の一つでもある。

最後に、本論文作成にあたり田中啓介助手には支援を受けた。また、討論における有益な助言を受けた。謹んで感謝する。

参考文献

[1] Sun Microsystems, Inc. : Sun Common Lisp User's Guide, Nov. 1988

[2] Franz Inc. : Allegro Common Lisp User Guide R. 3.0, Jun. 1988

[3] 湯浅太一、萩谷昌巳 : 京都コモンリスプレポート, 1985 年 9 月 10 日

[4] (株)CSK : 日本語 Common Windows マニュアル, 1989 年 8 月

[5] M.Ida, T.kosaka, K.Tanaka : YY on X, proc. of 2nd CLOS Workshop on OOPSLA '89, Oct. 1989